Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2021

Topic Notes: Matrix-Matrix Multiplication with MPI

We looked earlier at a simple program to multiply two matrices when we were looking at how to perform timings in a C program.

`https://github.com/SienaCSISParallelProcessing/matmult`

Here, we compute the product of two $n \times n$ matrices A and B in an $n \times n$ matrix C.

Recall that the value at each location in C is computed as the dot product of the corresponding row of A with the corresponding column of B.

Matrix-matrix multiplication can be parallelized pretty easily using threads. Since all threads will have access to the entirety of the matrices being multiplied as well as the result, it's just a matter of making sure that each entry in the result is computed by exactly one thread. We will see some examples of this later in the semester.

Matrix-matrix multiplication using message passing is not as straightforward:

- Since our memory is not shared, which processes have copies of the matrices?
- Where does the data start out? Where do we want the answer to be in the end?
- How much data do we replicate?
- What are appropriate MPI calls to make all this happen?

We will look at a few ways to accomplish this. Examples are in

`https://github.com/SienaCSISParallelProcessing/mpimatmult`

The MPI version of Conway's Game of Life used a distributed data structure. Each process maintains its own subset of the computational domain, in this case just a number of rows of the grid. Other processes do not know about the data on a given process. Only that data that is needed to compute the next generation, a one-cell overlap, is exchanged between iterations.

Think about that – no individual process has all of the information about the computation. It only works because all processes are cooperating.

The "slice by slice" method of distributing the grid was chosen for its simplicity of implementation, both in the determination of what processes are given what rows, and the straightforward communication patterns that can be used to exchange boundary data. We could partition in more complicated patterns, but there would be extra design and coding work involved.

The possiblities for a message-passing parallelization of the matrix-matrix multiply are numerous. Now the absolute easiest way to do it would be to distribute the matrix `A` by rows, have `B` replicated everywhere, and then have `C` distributed by rows. If we distributed our matrices this way in the first place, everything is simple.

`matmult_mpi_toosimple`

This program has very little MPI communication. This is by design, as we distributed our matrices so that each process would have exactly what it needs.

Unfortunately, this is not likely to be especially useful. More likely, we will want all three matrices distributed the same way.

To make the situation more realistic, but still straightforward, let's assume that our initial matrices `A` and `B` are distributed by rows, in the same fashion as the Life simulator. Further, the result matrix `C` is also to be distributed by rows.

The process that owns each row will do the computation for that row. What information does each process have locally? What information will it need to request from other processes?

Matrix multiplication is a pretty "dense" operation, and we to send all the columns of `B` to all processes.

`matmult_mpi_simple`

Note that we only initialize rows of `B` on one process, but since it's all needed on every process, we need to broadcast those rows.

Can we do better? Can we get away without storing all of `B` on each process? We know we need to send it, but can we do all the computation that needs each row before continuing on to the next?

`matmult_mpi_better`

Yes, all we had to do was rearrange the loops that do the actual computation of the entries of `C`. We can broadcast each row, use it for everything it needs to be used for, then we move on. We save memory!

Even though we do the exact same amount of communication, our memory usage per process goes from $O(n^2)$ to $O(\frac{n^2}{p})$.

Finally, we enhance the previous version to have the distributed result `C` written to a file.

`matmult_mpi_printans`

Like we saw in the MPI Game of Life, we use barriers to coordinate the writing of the file, with each process writing its own rows of `C` when it's turn comes up.